REMARKS

Claims 1-20 remain for further consideration. The rejections shall be taken up in the order presented in the Official Action.

3. Claims 1-5, 7-15 and 17-20 currently stand rejected under 35 U.S.C. §103 as allegedly being unpatentable in view of U.S. Patent 5,584,035 to Duggan et al. (hereinafter "Duggan"), further in view of the printed publication entitled "Cyberdog Could Be A Breakthrough If It's Kept On A Leash" by Henry Norr, published in Macweek, Vol. 8, Number 45, pg. 50, November 14, 1994 (hereinafter "Norr").

We shall first discuss the present invention, followed by a summary of the teaching of the cited prior art, and then the differences between the present invention and the cited art.

I. THE PRESENT INVENTION

The present invention is directed to an extensible and replaceable network-oriented component system that provides a platform for developing *network navigation components* that operate on a variety of hardware and software computer systems. Such a highly-modular cooperating layered-arrangement between the network component system and the component architecture layer allows a component to be easily replaced, and allows new components to be added. Significantly, the present invention provides a network navigating service which employs a *"component-based"* approach to browsing and retrieving network-oriented information as opposed to the monolithic application-based approach of prior browsing systems.

To provide the extensible and replaceable network *component* system of the present invention, the system includes a network *component* layer 450 (see Fig. 4) that delivers services and facilitates development of navigation *components* directed to computer networks, such as the Internet. Specifically, the network *component* layer 450 extends the functionality of the underlying *component* architecture layer 430 by defining network-oriented *components* 480 (see Fig. 4). The network *component* layer 450 and the *component* architecture layer 430 cooperate to provide a user with the ability to extend or replace, any

of the *components* of the layered computing arrangements 400 with a different *component* to provide the user with customized computer *network-related* services.

II. THE CITED ART

Duggan discloses a object based system comprising a user interface permitting manipulation of objects by users.

Norr briefly reports on a rumored product development by Apple Computer referred to as "Cyberdog" which is a suite of OpenDoc components with networking and communications capabilities, including an integrated set of Internet browsing tools. As reported by Norr, the object of Cyberdog is to enable a user to locate information anywhere on the global network, and embed extracts, together with hypertext links to sources of additional detail or background, directly into reports and presentations.

III. DIFFERENCE BETWEEN THE PRESENT INVENTION AND THE CITED ART

Claim 1 of the present invention recites an extensible and replaceable layered component computing arrangement which includes:

> "a software component architecture layer interfacing with an operating system to control the operations of the computer, the software component architecture layer defining a plurality of computing components;" (emphasis added, cl. 1).

In contrast, Duggan merely discloses the use of an application architecture, rather than the claimed *component* architecture. Notably, Duggan neither discloses nor suggests *"a software component architecture ..., the software component architecture layer defining a plurality of computing components"* (emphasis added, cl. 1).

The present invention provides a network navigating service which employs a "component based" approach to browsing and retrieving network oriented information as opposed to the monolithic application based approach of prior browsing systems. Specifically, the present invention is directed to a component based architecture, while Duggan is directed to an application based architecture. Referring to Fig. 1 of Duggan,

applications 22 interface with the operating system 12, via the object manager 16 and window interface. This figure merely illustrates applications; there is no illustration or even a suggestion of components. In contrast, Fig. 4 of the present invention illustrates a plurality of network components 480 cooperating with the network component layer 450 and the component architecture layer 430.

Attached as Exhibit A is portion of a printed publication which describes component software, and how the software industry is moving away from enormous monolithic applications (see last ¶, pg. 16). Exhibit B also includes a portion of a printed publication which briefly describes how component software expanded the power of object-oriented programming.

Although Duggan is directed to an object oriented system, the architecture is application based rather than component based (see Fig. 1 of Duggan, and col. 5, lines 21-24). A fair and proper reading of Duggan neither discloses nor suggests a component based architecture. It is alleged that *"Duggan et al. discloses a software component layer (object software 24) coupled to an operating system and defining a plurality of computing components (objects) [column 5, line 25 through [] column. 6, line 4]."* (Official Action pg. 5). However, this statement is factually incorrect since the use of objects does not imply the use of a component architecture.

In addition, neither Duggan nor Norr provides a suggestion that would lead a skilled person to combine these references. Duggan simply discloses linking of objects; it provides no teaching of a component architecture. Norr briefly reports on the rumored development of an Apple Computer product referred to as "Cyberdog", which is described as a suite of OpenDoc components with networking and communications capabilities, including an integrated set of Internet browsing tools. Assuming that Norr is a prior art reference, a skilled person working in the field of Internet browsing tools with a component architecture would not look to linking objects in an application architecture (i.e., Duggan). Although the Commissioner suggests that [the structure in the primary prior art reference] could readily be modified to form the [claimed] structure, '[t]he mere fact that the prior art could be so modified would not have made the modification obvious unless the prior art suggested the

desirability of the modification.'" In re Laskowski, 10 U.S.P.Q.2d 1397, 1398 (Fed. Cir. 1989), citing In re Gordon, 221 U.S.P.Q. 1125, 1127 (Fed. Cir. 1984). It is respectfully submitted that the combined teachings of Duggan and Norr fails to provide a proper suggestion that would lead a skilled person to arrive at the claimed invention.

Furthermore, it is evident that Norr is not prior art since Norr admits that he is merely reporting on rumored work at Apple Computer by stating *"[i]f I understand the secret project correctly . . ."* (pg. 1 of the reproduced article, emphasis added). One's own work, whatever the form of the disclosure, may not be prior art against one-self, absent a statutory bar.

With respect to claim 2, it is recognized that Duggan does not explicitly disclose a computing arrangement wherein the network navigation components are objects and the network component layer comprises application program interfaces delivered in the form of objects in a class hierarchy. It is then alleged that Norr discloses these elements and a skilled person would modify Duggan to include these features. However, the combination of Duggan and Norr fails to provide a proper teaching of several other claim elements. Specifically, as set forth above, Duggan does not provide a teaching of the network *component* layer or the software *component* architecture layer. In fact, Duggan provides no teaching of a *component* architecture.

It is also alleged that Norr discloses a network component layer (see Official Action, pg. 5, last paragraph). However, a fair and proper reading of Norr fails to reveal any suggestion of a network component layer as claimed. Norr merely makes a statement that the rumored Apple Computer product may include a suite of OpenDoc™ components with networking and communications capabilities. It *never states how* these capabilities will be provided.

Independent claim 7 is directed to a layered component computing arrangement which is patentable for at least all the same reasons as set forth above.

Independent claim 17 recites a method for providing network information services to a user of a computer system coupled to network computers. This method includes the steps of:

controlling operations of the computer system with an operating system coupled to a software component architecture of the system;

integrating a network component layer of the computer system with the software component architecture to provide a cooperating component computing arrangement;

creating a plurality of network navigation components using the cooperating component computing arrangement; and

invoking a selected one of the created network navigation components to provide a selected network service that enhances the ability of the user to access information on the computer networks.

As set forth above, Duggan fails to either disclose or suggest a component software architecture.

Claims 3-5, 8-15 and 18-20 are also patentable for all the same reasons as set forth above.

4. Claims 6 and 16 currently stand rejected under 35 U.S.C. §103 as allegedly being unpatentable over Duggan, in view of Norr and further in view of the printed publication entitled "Object Component Suites: The Whole Is Greater Than The Parts", Datamation, February 15, 1995; Vol. 41, Number 3, pg. 44 by Harkey et al. (hereinafter "Harkey").

Harkey discloses that distributed objects need to be packaged as components in order to properly build to order entire information systems by assembling off-the-shelf object components. Notably, Harkey discloses that an OpenDoc document will act as a central integration point for multiple sources of data that reside on different servers. For example, it is disclosed that end users will be able to create custom applications by choosing a container and populating it with active parts that live on servers anywhere on the network. It is imagined that the user will be able to access multiple data sources and business objects through multiple client/server connections from within a single visual container or document.

Claims 6 and 16 are patentable for all the reasons set forth above. Notably, Duggan provides no teaching of a component based system architecture. Hence, there is no proper

suggestion in either Duggan, Norr or Harkey that would lead a skilled person to combine the teachings of these references as suggested in the Official Action.

For all the foregoing reasons, reconsideration and allowance of claims 1-20 is respectfully requested.

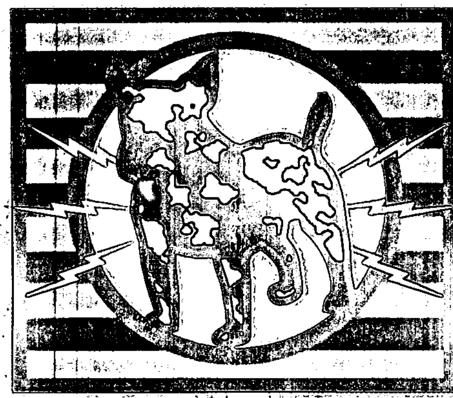
If a telephone interview could assist in the prosecution of this application, please call the undersigned attorney.

The Commissioner is hereby authorized to charge any other fees under 37 C.F.R. §1.16 and 1.17 that may be required, or credit any overpayment, to our Deposit Account No. 03-1237.

Respectfully submitted,

Patrick O'Shea

Patrick J. O'Shea
Reg. No. 35,305
Cesari and McKenna, LLP
30 Rowes Wharf
Boston, MA 02110
(617) 951-2500



Cyberdog™

The Complete Guide to Apple's Internet Productivity Technology

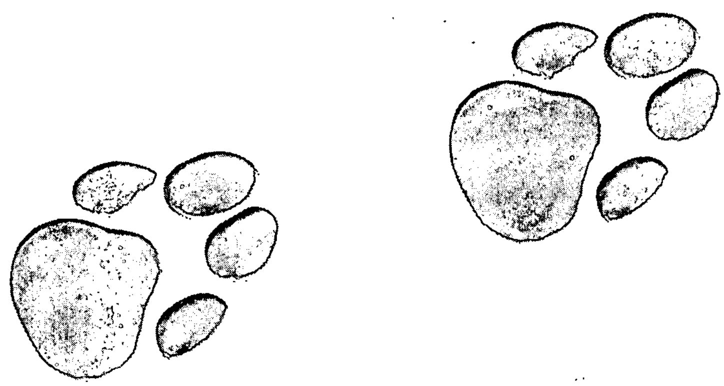
Exhibit A

Jesse Feiler

Cyberdog™

The Complete Guide to Apple's Internet Productivity Technology

Jesse Feiler



AP PROFESSIONAL

AP Professional is a division of Academic Press, Inc.

Boston San Diego New York
London Sydney Tokyo Toronto

AP PROFESSIONAL
An Imprint of ACADEMIC PRESS, INC.
A Division of HARCOURT BRACE & COMPANY

ORDERS (USA and Canada): 1-800-3131-APP or APP@ACAD.COM
AP Professional Orders: 6277 Sea Harbor Dr., Orlando, FL 32821-9816

Europe/Middle East/Africa: 0-11-44 (0) 181-300-3322
Orders: AP Professional 24–28 Oval Rd., London NW1 7DX

Japan/Korea: 03-3234-3911-5
Orders: Harcourt Brace Japan, Inc., Ichibunan

Australia: 02-517-8999
Orders: Harcourt Brace & Co. Australia, Locked Bag 16, Marrickville, NSW 2204, Australia

Other International: (407) 345-3800
AP Professional Orders: 6277 Sea Harbor Dr., Orlando FL 32821-9816

Editorial: 1300 Boylston St., Chestnut Hill, MA 02167; (617) 232-0500

Web: http://www.apnet.com/

This book is printed on acid-free paper. ♾

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24–28 Oval Road, London NW1 7DX

ISBN 0-12-251330-4

Printed in the United States of America
96 97 98 99 IP 9 8 7 6 5 4 3 2 1

Preface

Cyberdog 1.0 was released by Apple on May 12, 1996—just in time for Apple's annual World Wide Developers Conference. Five years earlier, in May of 1991, Apple released the last major upgrade to its operating system—System 7; that release also coincided with a meeting of the World Wide Developers Conference.

The mechanics of the two releases could not be more different. In the case of System 7, great moving walls of San Jose's Convention Center were drawn up to the accompaniment of music and flashing lights, revealing trucks loaded with shrink-wrapped cartons containing the diskettes that held the new operating system. The 4,000 enthusiastic developers crowded around, grabbing the long-awaited software and carrying it off so that they could install it as quickly as possible on their computers back at home or in their offices. Some even rushed to a desk in the lobby to send the precious parcels by overnight courier to waiting friends and colleagues.

Five years later, Cyberdog 1.0 was released via the Internet. On the opening day of the Conference, an announcement

was made that Cyberdog was released; there were no lights, no shrink-wrapped cartons, and no pushing and shoving. Developers were invited to download it from cyberdog.apple.com; no overnight packages were shipped to faraway places. Many developers returned to their hotels and downloaded the software onto PowerBooks, installing it before the lunch break was over. Some thirty thousand copies were downloaded within a few weeks; within a month the rate had settled down to 1,000 downloads a day.

The morning keynote speeches (by Apple CEO Dr. Gil Amelio and Chief Scientist Larry Tesler) were awaited by people around the world. They didn't have long to wait; the sessions were broadcast on the World Wide Web—some with audio and slides, others as full video. Articles in the press analyzing the speeches began to appear on Web sites late in the day; mailing lists and news groups started to buzz with comments and critiques as soon as the lights went down on the stage.

This is the world of the Internet; this is Cyberdog's world. It's a new and rapidly changing world in which communication—rich communication involving far more than text—is nearly instantaneous and taken for granted.

A Note on Versions

The illustrations and examples in this book are taken from Cyberdog 1.0. With the advent of the Internet as a distribution medium for software, major releases of software (often at intervals of a year or more) are becoming a thing of the past. Development cycles are condensed, with the ease of releasing a continuing series of minor revisions providing constantly improving software.

You may well have a version of Cyberdog that is later than 1.0; you may have third-party replacement parts for the

standard Cyberdog parts. In any event, the structure and basic behavior of Cyberdog is as it is described here.

Accessing the Internet

There are two basic types of Internet connections: hard-wired connections made through a network operated by an organization such as a school, corporation, or other entity; and connections made by an individual dialing up an Internet Service Provider (ISP). In this book, you will find detailed descriptions of the second type of access (generally described as SLIP or PPP access). Access through a network which is itself connected to the Internet is discussed in less detail.

The reason for this is that if you are connecting to the Internet through a local network, there must be someone fairly technical around who maintains your network and its computers. If you need to deal with such esoterica as net masks and router numbers, such a person will provide you with the installation-specific details that you need. He or she need not know anything about Cyberdog, and you need not know anything about all that network stuff in order to make your network connection work.

"Cyberdog-savvy"

Cyberdog is implemented using the OpenDoc component software architecture. There is no monolithic Cyberdog application to launch—the necessary software is loaded as needed. OpenDoc documents (including those created and maintained by Cyberdog) consist of OpenDoc parts. A part is said to be Cyberdog-savvy if it knows about Cyberdog—which in general then means that whenever you are using a

Cyberdog-savvy part, you have access to all of Cyberdog's services. All this will be explained in due time. For now, just remember that "Cyberdog-savvy" means exactly what you think it does.

Acknowledgments

Many people have provided help in bringing this book to fruition. Their assistance has been invaluable and has made this book possible. Comments and suggestions have improved the manuscript; however, any remaining errors, omissions, or infelicitous phrases must be laid (like a rain-soaked newspaper) at the author's doorstep.

At AP PROFESSIONAL, Chuck Glaser (who initiated this book), Jeff Pepper, Ken Morton, Mike Williams, and Cindy Kogut have been a joy to work with. Andrew Rubenfeld's copyediting has made the book more readable than it would otherwise have been, and Owen Hartnett's comments on technical issues have been very useful.

At Apple, Glenn Fisher has provided invaluable help on a variety of questions large, small, obscure, and bizarre.

Jim Black from Apple Evangelism has once again provided support and assistance. It's possible to write a book about Apple's most exciting technologies without Jim's help—but it's a lot easier when he's on the project.

Barbara Butler once again has provided insightful comments on the manuscript. Carole McClendon (as always) has been helpful and supportive in shepherding the project along.

Introduction

When the Brooklyn Bridge was opened in 1883, crowds of people turned out to see this "eighth wonder of the world." They marveled at its suspension cables, strung from miles of steel cabling spun at the Roebling Iron Works; they wondered at the towers that rose above New York's East River from which the cables were suspended, carrying a roadway that spanned a distance of 1,595 feet—the longest suspension bridge in the world at that time. They experienced the odd sensation of walking from the City of Brooklyn to the City of New York (which in those days were two separate cities) rather than taking a ferry.

The discussions that had gone on throughout both cities since the start of construction in 1867 now reached a fever pitch. Everyone agreed that the bridge would have an enormous impact on the lives of many—if not all—residents. Clearly, there was money to be made as a result of the bridge's opening: profits not just for the bridge's railway and trolley operators, but also large gains by real estate de-

velopers who owned property in the far reaches of Brooklyn (which suddenly weren't quite so distant from Manhattan after all). Beyond these, other people saw opportunities for new endeavors and adventures made possible—or even necessary—by the Great Bridge.

Not all the consequences of the bridge's opening were positive. There was harm to the owners of the ferries that plied the East River (although the Fulton Ferry lasted until 1924). Concern was raised that the bridge would allow sinners and miscreants from Manhattan to flood into Brooklyn, which after all was known as the "City of Churches." (Oddly enough, a corresponding fear of an influx of godliness from Brooklyn to Manhattan was not reported.)

Over time, the bridge became an everyday part of the city of New York (which was consolidated with Brooklyn and the other boroughs in 1898). Fewer and fewer people came solely to gawk at the bridge; more and more came simply to use it—to get to their destinations. Even today, there are people who come just to see the Great Bridge—to stroll its promenade with the breathtaking views of New York's harbor and skyline—but there are millions who use the bridge routinely. Riding in buses and cars with their eyes glued to their newspapers and books, they may not even notice their crossing of the East River.

Sightseeing on the Internet

A century later, the Internet is very much like the Brooklyn Bridge. While the Net has been around for a good quarter of a century, the World Wide Web, which is what has made its use feasible for scores of millions of people around the world, is only a few years old. The ratio of gawkers to commuters is similar to what it was in the early days of the Brooklyn Bridge. The opportunities for profit and loss that were apparent with the bridge are there on the Internet.

Like the unforeseen consequences of the opening of the bridge, many uses of the Internet also lie in wait. Even the fear of the invasion of sin has its parallels today.

The Internet as a new and exciting wonder of the world attracts as much attention as the Brooklyn Bridge did in its time. With the passage of time, the Internet—like the bridge—will become ordinary and a useful part of everyday life rather than simply being a tourist destination.

Today, however, the tourist buses are pulling up in droves at the Internet. Software that can be used on the Internet sells like hotcakes—as do books about the Internet, and all sorts of products that can be marketed with an Internet "hook."

Making the Internet Useful

The evolution of the Internet to become an ordinary and useful part of our lives requires a new kind of software. Internet browsers, e-mail programs, and the like are great for tourists who want to experience the Internet as Internet; however, for people who want to use the Internet in a sophisticated and useful way, these programs are leading in the wrong direction. They perpetuate the notion that the Internet is something different and special rather than that it is simply a part (albeit extraordinarily powerful) of the computing world of the early twenty-first century. Only by integrating the Internet into the work and play that we do on their computers can we stop being sightseers and start being productive.

Wouldn't you know that the folks at Apple (who already have spent years and years thinking about how people use computers) would spend more time thinking about how people use the Net. The outcome of this thinking—and research, and development, and testing—is Cyberdog. Cyber-

dog is the first approach to the Internet that holds the promise of making this extraordinary technology ordinary.

It seems as if every company in the world is hell-bent on creating a super World Wide Web browser, a "killer" e-mail application, or the last word in Internet search engines. That's not what Apple is doing with Cyberdog. In fact, the Cyberdog engineers really don't want to hear you say, "Wow, what a cool Web browser." Cyberdog's browser is indeed pretty cool, but that's not what the engineers were looking for. What they want to hear is, "Oh, was I on the Internet? I was just writing a letter to my mom."

Making the Internet ordinary and thus useful is the goal of Cyberdog. And making Cyberdog ordinary and useful is the goal of this book. But don't make the mistake of thinking that "ordinary" means "dull." Making the Internet ordinary and useful is one of the most exciting (and needed) achievements of our time.

It's a little like what the first ten years of the Mac OS have been. Our colleagues in the DOS and Windows world have all those special-purpose programs to help them manage the complexity of their operating environment. They have those mysterious CONFIG.SYS and AUTOEXEC.BAT files along with a whole host of weird cables, cards, and dip-switches. We have the Mac.

Similarly, our colleagues who are mired in the details of the Internet have their browsers and e-mail programs and FTP programs and search engines that are designed to manage the complexity of the Internet. We have Cyberdog.

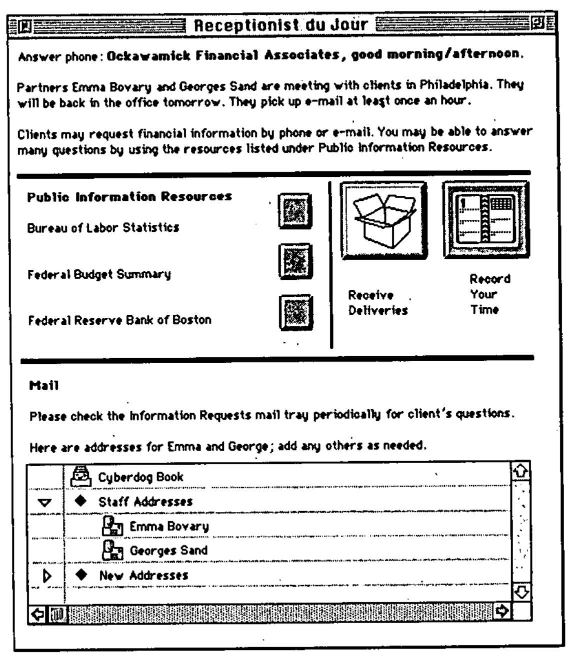
Some people have said that Apple's goal with the Mac OS is to make the computer disappear: you deal directly with what interests you without the distraction of dealing with some machine. In a similar spirit, you could say that Cyberdog lets the Internet disappear. We still have its vast resources, but we deal directly with them, without the

intermediation of UNIX code, idiosyncratic Net-only software, or our traditional applications that become stymied when forced to deal with the world of the Internet.

An Integrated Internet Solution

Even if you have used the Internet for years, it is quite possible that you have never seen a truly integrated Internet solution—one where the focus is on what you want to do rather than on the network and its specific tools. Here's a sneak preview of one of the examples shown in Part III of this book. It's called "Receptionist du Jour" (Figure 1).

FIGURE 1. Receptionist du Jour



The scenario is simple. A company hires a temporary receptionist to cover the office when both partners have to be out

of the office. On a typical day, the receptionist finds a document like that shown in Figure 1 on the computer.

Most people don't have much trouble figuring out what this is all about. Of the five buttons in the center of the window, it's pretty clear which one to click if you want information about the unemployment rate and which one to click when a courier rings the bell.

What isn't clear—and there's no reason why it should be—is which button automatically opens a World Wide Web browser in its own window with certain information in it, which button opens a search engine, and which one retrieves a text file from the Internet. Another button opens an e-mail message to be sent automatically with certain information filled in by the receptionist, and the last one opens a form to be filled in and stored on the computer's hard disk.

What is clear is what matters; what is hidden is what doesn't. That's the beauty of this technology.

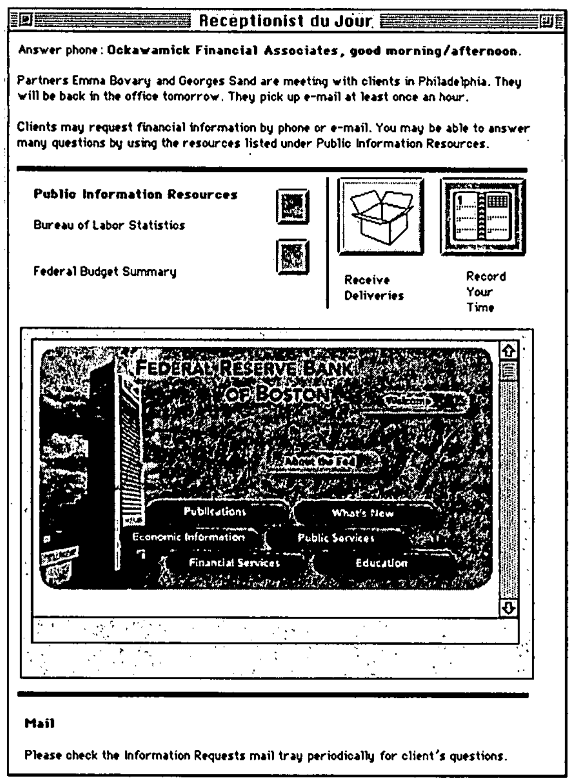
Constructing this document is extraordinarily simple; a step-by-step walk-through is provided in Part III. But just to whet your appetite, take a look at what one receptionist might do in some spare time.

Realizing that the Federal Reserve Bank of Boston Web site was the one that had most of the answers to clients' questions, this receptionist decided that instead of clicking on the button each time a question needed to be answered, maybe the Web site should be visible at all times in a revised Receptionist du Jour document (such as the one shown in Figure 2).

Everything's still live—the buttons at the top, the Web browser and the hot spots on its image, and the mail icons that are scrolled out of view at the bottom of the window.

The possibilities are endless, because Cyberdog lets you deal with the information and issues that you care about. The mess of Internet terminology and protocols as well as the complexity of complicated applications are gone.

FIGURE 2. Receptionist du Jour with Embedded Browser



If you've never used the Internet before, this may all seem quite obvious to you. If you're an Internet veteran, though, you may be looking for the browser, the e-mail application, the Gopher or TELNET applications. They're gone—subsumed in the flexible, task-centered architecture of Cyberdog.

Cyberdog is described in Part I of this book. You'll find chapters on its terminology—as well as the basic terminology of the Internet and of OpenDoc.

Part II tells you how to use Cyberdog and the basic tools it comes with. In Part III, you'll find a number of real-life solutions built around Cyberdog (including the Receptionist du Jour example previewed here).

Finally, Part IV covers the Cyberdog API. You don't need to be a programmer to use Cyberdog, and you can create very powerful Cyberdog solutions without writing a line of code. Nevertheless, if you *do* want to write code, you can expand and extend Cyberdog in many ways.

I Cyberdog Basics

1 Why Cyberdog?

In 1984, purchasers of the first Macintosh computers were able to take them home, plug them in, and start using them immediately. This was due not only to the easy-to-use operating system, but also to the fact that each machine came bundled with all the software one would ever need.

When you wanted to write a letter, you popped the MacWrite disk into the computer; finished with that, you could drag the MacWrite disk to the Trash (no arcane commands here), and slide in the MacPaint disk so that you could draw a picture. That was it—and to some, it seemed like the last word in computing.

The Landscape of Computing

Of course, far from being the last word in computing, it was little more than preliminary throat clearing. Today, the landscape of computing is far from the simple world of 1984. Today's picture is much bigger, and far more complicated than it was over a decade ago. In the last few years,

with the widespread use of networked computers in organizations large and small, and with the phenomenally rapid growth of the Internet that links individuals, organizations, and their networks together, the picture has changed dramatically.

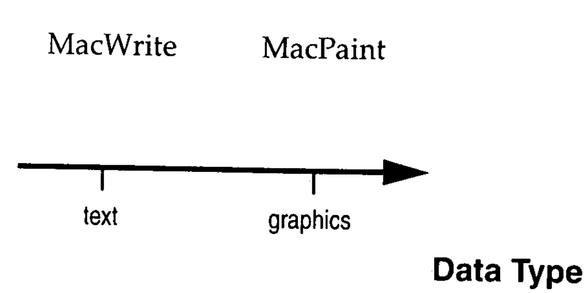
With each new feature and functionality that is added to those already in use comes added complexity. Worse, the interplay among new and old features and functionalities presents an exponentially rising curve of difficulty for the hapless user who wants to use any features in this intricately connected world. While each single new advance is welcome in and of itself, the chaos that results as they are integrated (or not integrated) into a person's life starts to decrease productivity, increase frustration, and make people long for the good old days.

The history that follows is an over-simplification of what actually happened, which was far more complicated, and much worse. If you didn't actually live through it, you don't want to know the details.

Data Types

The beginning was very simple: text and graphics, and applications to modify them. Figure 1-1 shows this paradigm.

FIGURE 1-1. The First Dimension: Data Types

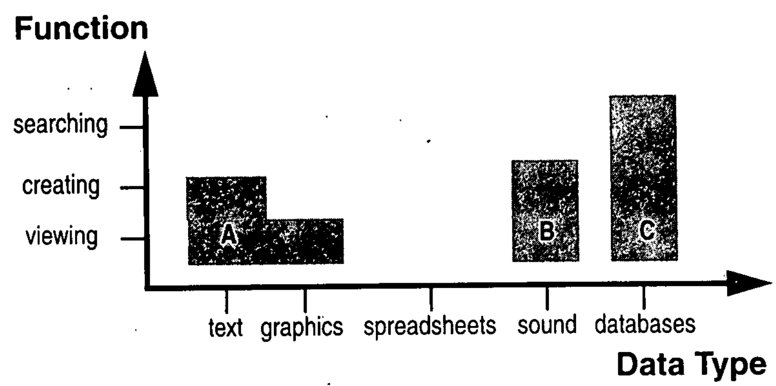


All data and applications were closely linked by data type. As new types were added (spreadsheets, for example), new applications arose. Before long, the Data Type dimension shown in Figure 1-1 had spreadsheets, sound, and databases as additional types that people could use.

Function

As time went on, more and more applications arrived on the scene; soon users had a choice among several different applications when they wanted to work on their data.The differentiation among applications that dealt with the same data types came about as they provided different functionalities for the user. You could do different things to the same data with different applications. The landscape of computing thus added a second dimension: Function.

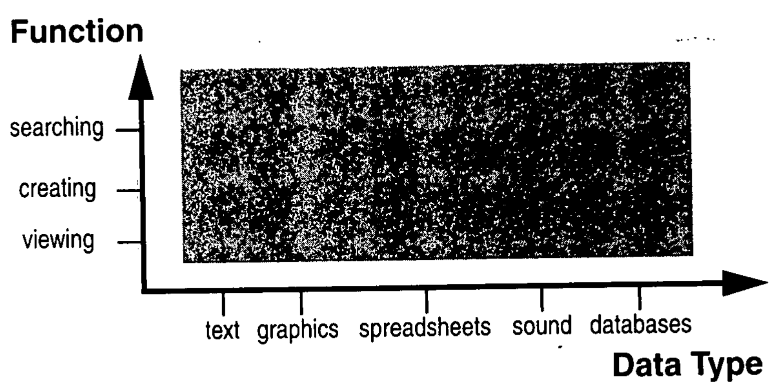
FIGURE 1-2. The Second Dimension: Function



In Figure 1-2 you can see new data types along the dimension of Data Types; in addition, the Function dimension is added, showing in a simplistic manner the sorts of things you could do to your data. For example, A represents a basic word processor that allows you to create and view text, and to view graphics that were created from another application and pasted in. B represents a simple sound program that lets you create and view (listen to) sounds. C shows an elementary database program that allows searching, creating, and viewing data in its own formats.

But what if you wanted to store graphics in your database? What if you needed to put a sound in your spreadsheet? In an increasingly competitive marketplace, software developers sought to develop applications that were all things to all people—suites of applications or "works" products that managed all functions for all data types equally well. Schematically, they were searching for the application diagrammed in Figure 1-3.

FIGURE 1-3. All-in-One Application



The problem with these monster applications was that they could never be revised fast enough to keep up with the new data types and new functions that were being invented by imaginative developers and users. Additionally, the internal functionality of these applications made their development, testing, and documentation a nightmare. Individual users suffered (usually in silence) as they managed enormous applications with frequent revisions—and which contained much code that they never cared about.

To be sure, the person who wanted to store text, graphics, and sound in a database could do so with an application such as the one shown schematically in Figure 1-3—but there was also all that extraneous code that supported spreadsheets. Users of major "productivity" applications

use a remarkably small amount of their programs' functionality: typically far less than half, often less than a third.

Furthermore, the neat categorization of data types and functions was under assault from people who did not see them starkly differentiated in the way that many software designers did. To most people, data types were a constantly changing continuum, and the primitive functions that made so much sense to designers weren't what people wanted to deal with.

Data Types and Functions as Continua

The view of highly compartmentalized data types and activities is very useful to both computer users and developers. In a world of large applications, the choice of your software is critical. Not only is there the cost of buying the software, but there is the much greater cost of entering and converting your data to that format—with the attendant risk of even higher costs if you have to switch to another application. If the world truly did contain these highly separate activities and data types, these choices would be easier.

Unfortunately, people just don't work this way (although many software developers like to pretend that they do).

Data Types. From a programmer's point of view, an image is totally different from the replacement text which describes the image when you can't see it. People, however, don't see such a tremendous difference. The old saw, "A picture is worth a thousand words," is testament to the fact that most people do indeed find a relationship between text and graphics—despite the fact that they have different file formats on the hard disk.

Functions. When it comes to the Functions dimension, we usually do only one thing at a time—search, or read, or write, etc. However, we rapidly switch our activities, and what might appear as a lengthy span of a single activity often reflects a multitude of different actions as we use the computer as a tool to carry out a complex human goal. The

rather simplistic functions that most of our software supports don't really matter to us. To systems analysts and developers, using a Personal Information Manager to look up an address is a different activity from writing a letter, which is in turn a different activity from faxing the letter to its recipient. To most people, however, these are all aspects of one activity: writing a letter.

Thus, in the real world we constantly move around the landscape of computing, rarely landing uniquely on a single point on an axis, but quickly moving among a number of functions to accomplish a humanly meaningful task, and looking at our data in various ways.

Software Implications. Now if the world of computer information is composed of continua rather than discrete boxes, and if it is true that we constantly switch our activities, that we usually can't be pigeon-holed into compartments with nice names on them, what does that mean for the software that we must use?

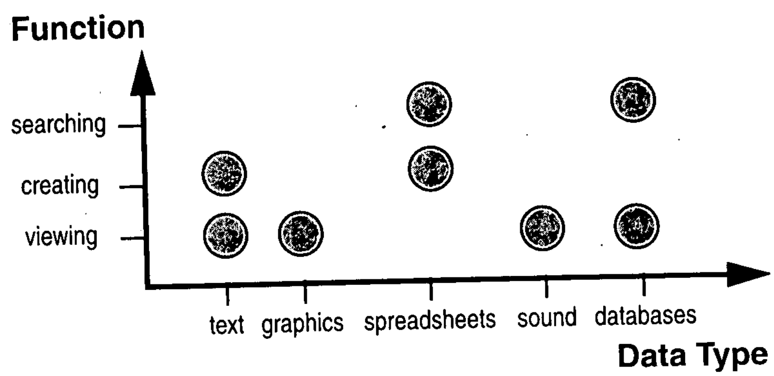
Most of the software that we are familiar with is designed for the comfortable (yet unreal) world with sharp distinctions between different categories of data and activities. This model doesn't support the flexibility that we require. Fortunately, a revolution in software in the mid-1990s is coming to the rescue.

Component Software

The computer industry is moving away from enormous monolithic applications to a new software architecture. In this new architecture, the software is assembled—often dynamically at run-time—that is needed to accomplish a specific task.

If you suddenly need to deal with video in the middle of a word processing document, you simply put a piece of video in, and the operating system hauls in the video support code for you. In the more traditional way of building applications, to put video into your word processing document, the word processor itself had to know about video. In fact, it—or your spreadsheet or database program—had to know about every kind of data which it ever possibly could handle. No wonder these programs became monstrous in size and required frequent updates. In the world of component software, a person might use the software components diagrammed in Figure 1-4, rather than the enormous block of code shown in Figure 1-3.

FIGURE 1-4. Component Software



Software components are logical, distinct pieces of functionality that can be combined and re-combined as needed. They are (in their ideal sense) independent of operating systems and languages—and often of one another. Components discussed in the context of this book (technically CORBA-compliant OpenDoc components) define their interfaces rigorously using an **Interface Definition Language (IDL)** that provides neutrality across many environments. As a result of this rigorous definition of interfaces, CORBA-compliant software components can interact with one an-

other across different address spaces, different operating systems, and different computers on a network.

Whereas traditional applications provide the functionality of software components together with the structure for managing and maintaining their operations and data, in the world of software components, those roles are performed by a neutral, high-level architecture that is implemented close to (or as part of) the operating system—independent of the components that it manages.

On the Mac OS (as on Windows, AIX, and OS/2), the most sophisticated and powerful way of managing component software is with OpenDoc, a technology pioneered by Apple, IBM, and Novell/WordPerfect.

OpenDoc. With OpenDoc, you typically work with a document shell that is a rather general and amorphous creature; into this shell you place parts that interest you—parts that may contain different data types such as text, graphics, sound, or video. The document shell needs to know almost nothing about the parts that are contained within it. As a result, instead of needing bigger and bigger applications to handle all the possible data types that exist, a simple document shell—conforming to the OpenDoc architecture—can let you embed all sorts of data in your document.

Chapter 4 provides a fuller introduction to OpenDoc.

Since OpenDoc components are typically relatively small, and since they are assembled dynamically as needed, OpenDoc is ideally suited to help you move dynamically through the landscape of computing where you do different tasks with different types of data—all in the service of helping you to achieve a single, meaningful goal.

Inter

As this monumental change was in the process of being assimilated by developers, designers, and computer users, something else was happening: the Internet.

Internet Arrives

With the Internet, people began sending e-mail to one another, easily transferring files, chatting in real-time, and connecting to remote computers in order to use them as easily as the machine on their own desk (or lap). They also started searching the immense information resources on the Internet, and began browsing the ever-expanding pages of the World Wide Web.

The Internet appeared in the early 1990s as a totally new creature to most people. They needed new software, and developers were happy to provide it for them.

The Internet was more than a quarter-century old by that time. Its "overnight success" mirrors that of the artist who struggles for years and years only to be discovered when the time is right. Chapter 3 provides more information about the Internet.

At first, Internet software was made to fit into the same model as that of the desktop software shown in Figure 1-2. There were some new Internet-only functions (e-mail, Gopher and Veronica search engines, file transfer protocols [FTP], etc.). There were some new data types as well—most significantly, HTML-based World Wide Web pages. These were all jammed into the same old model, and new Internet applications emerged, which quickly became as overblown and hard to manage as their desktop predecessors. People who wanted to use the Internet started dealing with an environment with many more data types and many more functions than they had had before. Even worse, the new

data types and new functions existed next to the previous data types and functions. Many people had a sense that something was wrong. Somehow or other the Internet didn't feel all that different from the desktop (despite the fact that experts proclaimed it so).

The Difference Between the Net and the Desktop

According to many pundits, using the Internet is a totally different kettle of fish from using your local computer. Sending e-mail to a colleague has nothing whatsoever to do with writing a memo to be mailed or faxed to that same colleague. It doesn't matter that the e-mail message and the memo may be absolutely identical, the activities are fundamentally different and require completely different software.

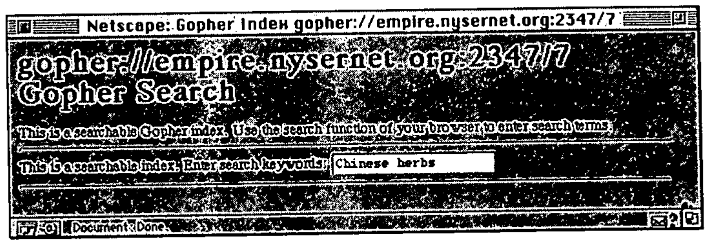
This, of course, is hogwash. But there's more. Some authorities say that you actually do different things on the Internet than you do on your desktop. And these differences are so profound that the tools that you use for one cannot be used for the other.

There is no doubt that there are some differences between using the Net and using your computer locally, but these are differences of degree. You search the Internet for information about something that interests you—perhaps Chinese herbs. You may well search for something that you've seen before and have just lost track of, or you may search blindly for any information that may be there. Using popular Internet search engines, you can search the text of Web pages or the titles of files that are posted to the Internet. Clearly (we are told) this is an activity that is unique to the Internet.

This one isn't hogwash; this one is balderdash. Look at the figures below.

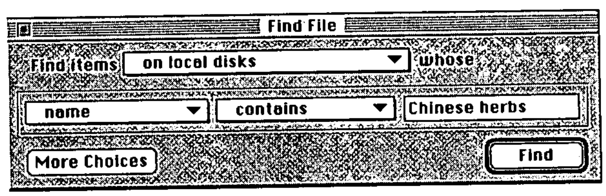
In Figure 1-5, you see the interface to a Gopher search engine—a tool for searching titles of articles on the Internet.

FIGURE 1-5. Gopher Internet Search Interface



This, of course, has nothing to do with the Find File dialog (Figure 1-6) that you probably use routinely to find files on your own computer. (Or does it?)

FIGURE 1-6. Find File Dialog



Human Activities Don't Change with the Net

Where is the tremendous difference between searching the titles of files on the Internet with Gopher and searching the titles of files on your computer with Find File?

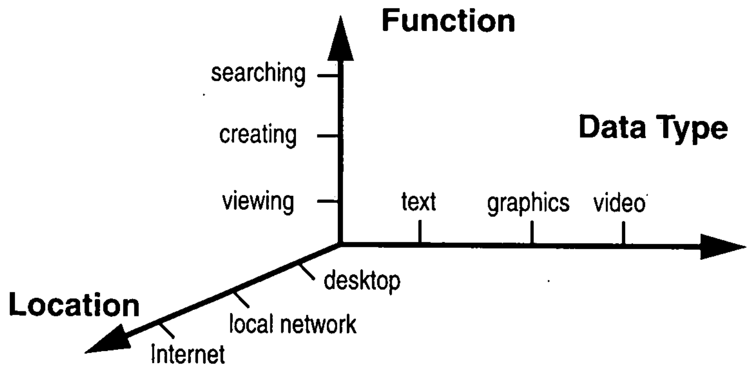
The activities that we regularly do—whether on the Internet or on a local computer—are basically similar. Some are more likely to be done locally; others are more probably done over the Net. But the fundamental activities are not so compartmentalized as many would have us believe.

The complexity of the two-dimensional model of functions and data types was getting to be too great. A simplification was in order: a third dimension was added to the picture.

A Three-Dimensional View of the Landscape of Computing

Imagine a three-dimensional view of the landscape of computing, as shown in Figure 1-7.

FIGURE 1-7. Three Dimensions of the Landscape of Computing



By adding the dimension of Location to the picture, things get a lot simpler. No longer do you have to consider Internet searching and desktop searching as two different functions—they are the same function, one performed on the desktop location and the other on the Internet.

Similarly, the parallel between e-mail and word processing becomes clearer with the addition of the Location dimension. In Figure 1-7, find the intersection of "creating," "text," and "desktop"—that's your local word processing program. "Creating," "text," and "Internet" comprise e-mail.

The use of traditional monolithic applications in this three-dimensional world is more problematic than ever. If the "all-things-to-all-people" application diagrammed in Figure 1-3 is a nightmare to develop, test, learn, and use, imaginge what its three-dimensional counterpart would be! In the three-dimensional landscape of computing, OpenDoc—with its component software architecture, cross-platform

and distributed support of objects, and simplified user experience—becomes imperative.

Which is where Cyberdog comes in.

Cyberdog

Cyberdog is a suite of OpenDoc parts from Apple. Just as there are OpenDoc parts that support different data types along the Data Type axis, there are OpenDoc parts that support different activities along the Function axis. Computer users as well as developers combine these parts as needed to accomplish their goals.

With Cyberdog, the three-dimensional landscape of computing is completed for OpenDoc users. Parts that provide connectivity to the Internet are available to be included in documents and solutions as easily as embedding a picture. The trauma that people went through in choosing the right word processor for their purposes was followed by the trauma that they went through in choosing the right Internet tools. Each decision was big and involved significant investments of time and money—with big problems if the wrong choices were made.

Now the choice is far less monumental. Since you are choosing only small OpenDoc parts, it's much easier to replace them. With Cyberdog parts providing Internet functionality on a small and specific basis, you don't have to worry that your choice of Internet software will one day turn around and cause you problems in another area. With OpenDoc, parts are remarkably independent; changing one rarely affects the others.

Summary

In the chapters that follow, you will find Cyberdog's basic design and terminology (Chapter 2), a brief introduction to the Internet (Chapter 3), and a basic guide to OpenDoc (Chapter 4).

The most important point to remember about the three-dimensional landscape of computing is that it represents the real world. We have been forced to pigeon-hole our activities and to conform our tasks to the software that is provided for mass markets. With the advent of OpenDoc, we have the opportunity to work in a more natural manner, with software molding itself to our wishes rather than the reverse.

And with Cyberdog added to the world of OpenDoc parts on the Mac OS, we can bring this more natural way of working to our travels through the Internet as well as on our desktops.

This is a remarkable advance and promises a major increase in productivity. And all at a remarkably low cost. All you really have to do is learn a little new terminology—which starts on the next page.

Exhibit B

SECOND EDITION

ALL NEW MATERIAL

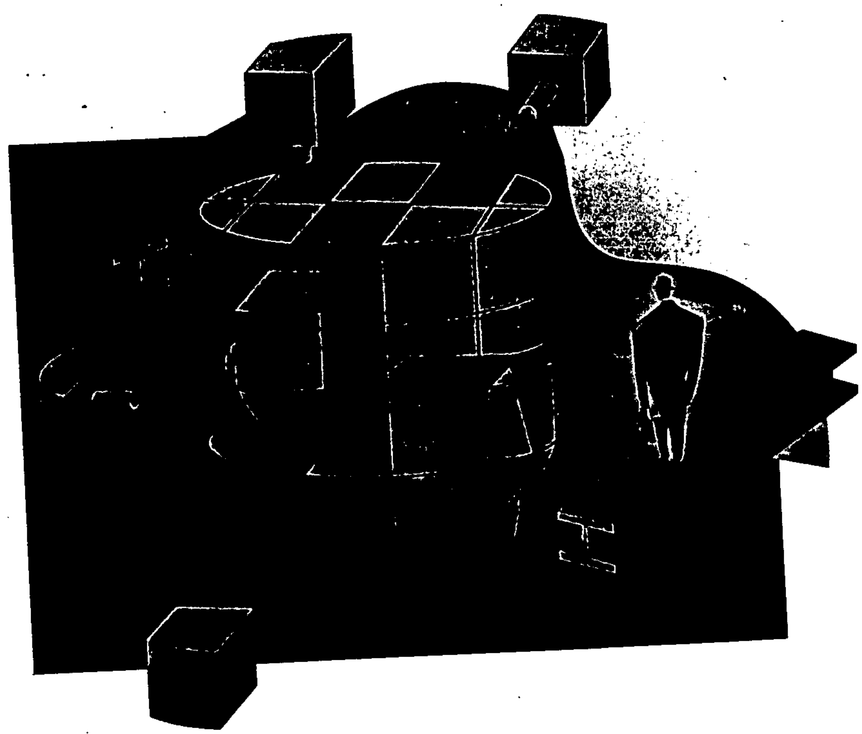
MICROSOFT PROGRAMMING SERIES

INSIDE OLE

SECOND EDITION

Includes source code files for over 75 programs

The Inside Track to Building Powerful 32-Bit Component Software with Microsoft's Object Technology



KRAIG BROCKSCHMIDT

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Brockschmidt, Kraig, 1968–
Inside OLE / Kraig Brockschmidt. -- 2nd ed.
p. cm.
Includes index.
ISBN 1-55615-843-2
1. Object-oriented programming (Computer science) 2. Microsoft Windows (Computer file) I. Title.
QA76.64.B76 1995
005.75--dc20 95-15969
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QMQM 0 9 8 7 6 5

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (206) 936-7329.

Acquisitions Editors: Dean Holmes and David Clark
Manuscript Editor: John Pierce
Project Editor: Kathleen Atkins
Technical Editor: Jim Fuchs

CHAPTER ONE

An Overview of OLE

All evolution in thought and conduct must at first appear as heresy and misconduct.
—George Bernard Shaw (1856–1950)

Many years from now, a Charles Darwin of computerdom might look back and wonder how the Microsoft Windows API (Application Programming Interface) evolved into an object-oriented operating system in which most of the "API" is provided through the objects. The technology known as OLE is the genesis of this transformation. It will change how you program—and eventually how you use—Windows. In the beginning, you'll probably regard it as utterly strange and difficult, no matter what your background. But don't feel too threatened. OLE doesn't ask you to throw away any knowledge you've accumulated. OLE preserves most of your knowledge and your code—extending the reach of both to help you reach new heights in software. OLE exists to facilitate and enable the integration of components, either parts of the system of applications or simply stand-alone elements, and it is the capacity for integration that creates a tremendous opportunity for innovation and growth.

Most people involved in the computer industry acknowledge the value of objects or object-oriented programming—whatever these things are. An object offers a certain set of features that define its functionality and the information it manages. Objects are powerful because a single pointer or other reference to an object provides access to all that object's functionality and content. Without objects, programs usually have to maintain multiple variables and pass them to isolated functions to achieve the same ends, adding greatly to complexity. By grouping such variables and functions in a single unit, an object, you create a layer of abstraction that simplifies many programming tasks and provides additional power. This abstraction is a key to extensibility, where new types added into a system can appear and act as existing types.

OLE's contribution to "objects" is threefold. First, OLE enables the integration of components—packages of one or more objects—that object-oriented languages such as C++ cannot, and in that way OLE strongly complements such languages. Second, OLE works independently of programming languages through a binary standard, allowing the oldest legacy code to integrate as a component alongside the newest object-oriented code. Furthermore, objects in OLE share the characteristics of all objects: they are encapsulated, they are polymorphic, and they are reusable. OLE is often denounced as heretical because of minor differences between OLE objects and other objects. OLE's language independence accounts for most of the differences. Such accusations fail to recognize the third category of OLE's tremendously powerful innovations—its concept of object "interfaces" and the idea that objects can have multiple interfaces, each of which represents support for a feature. These interfaces are, literally, different ways of communicating with an object—that is, accessing its functionality and its information. The abstraction of interfaces thus simplifies even more programming tasks, some heretofore impossible.

So while objects are very powerful because they factor functionality and information into tangible units, OLE takes it one step further and factors that functionality and information into related groups, which provides a way for the client or user of an object to ask the object—at run time—which features it supports. This arrangement makes it possible for the independent development and deployment of components over time, without requiring any recompilation of anything else in the system, without requiring a restart of the system, and without having to hassle with compatibility problems. OLE solves such problems, making evolutionary object-oriented component integration possible.

OLE is, at its core, a collection of component services accessed through objects that expose specific interfaces. You can customize some of these services with objects of your own that hook into this core, and if the services you want are not part of OLE, you can also create your own custom services using exactly the same mechanisms. In this sense, OLE is an extensible service architecture, in which a custom, or extended, service transparently becomes part of the core set of services and is immediately available to everything else in the system.

OLE is built on a foundation, the Component Object Model (COM) as it's called, that is both language independent and location independent. Language independence means that you can use any language to implement components as long as the programming tool in whatever language supports

the idea of an OLE interface. With OLE, you can encapsulate an existing piece of code as an object without having to necessarily rewrite that code. Location independence means that these components and objects can be implemented and shared in separate dynamic link libraries (DLLs), or executables (EXEs) on the same machine or across machines.[1]

After defining and exploring exactly what OLE is, we'll take a look at OLE's concepts of components, objects, and interfaces. We'll then briefly examine each specific OLE feature (or technology), describing how your software might take advantage of it (that is, profit from it). By using these features today, you can begin to transform your software to take advantage more readily of the future evolution of Windows (that is, profit from it tomorrow), which is going to involve not only today's OLE but also evolutionary enhancements that Microsoft—and others—will be adding in the future. OLE might seem like heresy today. But the fact is that OLE is a powerful evolution of "objects," and someday Darwin's successor will have plenty to say about OLE—the origin of a new species of incredibly sophisticated and powerful software.

What Is OLE?

Because this hefty tome you're holding is all about OLE, it behooves us to describe exactly what OLE is. To set the stage, we need to understand a little about the programming problems that OLE designers intend to solve.

History: Why OLE?

Constant innovation in computing hardware and software have made a multitude of powerful and sophisticated applications available to users at their desktops and across their networks. Yet, with such sophistication have come many problems for developers, software vendors, and users. For one, such large and complex software is difficult and time-consuming to develop, maintain, and revise. Revision is a major problem for monolithic applications, even operating systems, in which features are so intertwined that they cannot be individually and independently updated or replaced. Furthermore, software is not easily integrated when written using different programming languages and when running in separate processes or on separate machines.

1. At the time of writing, cross-machine OLE is not yet available, although the OLE architecture is designed to anticipate such capabilities.

Even when integration facilities have been available, the programming models for working with different services across various boundaries have not been consistent. The trends of hardware downsizing and greater software complexity are driving the need for distributed component environments. This requires a generic set of facilities for finding and using services (components), regardless of who provides them or where those services run, as well as a robust method for evolving services independently over time without losing compatibility with clients of earlier versions. Any real solution to these problems must also take advantage of object-oriented concepts and be capable of working with legacy code—that is, look to the future without forgetting history.

As an example, consider the problem of creating a system service API that works with multiple providers of some service in a polymorphic fashion. In other words, you want a client of the service to be able to transparently use any particular provider of the service without any special knowledge of which specific provider—or implementation—is in use. In traditional systems, every application calls a central piece of code to access meta-operations such as selecting a service and connecting to it. Usually this code is a service, or an *object manager,* that involves function-call programming models with system-provided handles as the means for object selection. But once applications have used the object manager to connect to a service, the object manager only gets in the way like a big brick wall and forces unnecessary overhead. Yuck.

Worse yet, such traditional service models make it nearly impossible for the service provider to express new, enhanced, or unique capabilities to potential clients in a uniform fashion. A well-designed traditional service architecture, such as Microsoft's Open Database Connectivity (ODBC) API, might provide the notion of different levels of service. Applications can count on the minimum level of service and then determine at run time whether the provider supports higher levels of service in certain predefined quanta. The providers, however, are restricted to providing the levels of services defined at the outset by the API; they cannot readily provide a new capability that clients could discover at run time and access as if it were part of the original specification. To take the ODBC example, the vendor of a database provider intent on doing more than current ODBC standards permit must convince Microsoft to revise ODBC in a way that exposes that vendor's extra capabilities. In addition, the Microsoft bottleneck limits the ability for multivendor initiatives independent from Microsoft to exploit an existing technology for their own purposes. Thus, traditional service architectures cannot be readily extended or supplemented in a decentralized fashion—you have to go through the operating system vendor. Yuck.

Traditional service architectures also tend to be limited in their version handling. The problem with versioning is one of representing capabilities (what a piece of code can do) and identity (what a piece of code is) in an interrelated, ambiguous way. A later version of some piece of code, such as "Code version 2," indicates that it is like "Code version 1" but different in some distinct and identifiable way. The problem with traditional versioning in this manner is that it's difficult for code to indicate exactly *how* it differs from a previous version and, worse yet, for clients of that code to react appropriately to new versions—or to not react at all if they expect only the previous version. The versioning problem can be reasonably managed in a traditional system when there is only a single provider of a certain kind of service. In this case, the version number of the service is checked when the client binds to the service. The service is extended only in an upward-compatible manner (a significant restriction as software evolves over time) so that a version *n* provider will work with consumers of versions 1 through *n*–1 as well, and references to a running instance of the service are not freely passed around by clients, all of whom might expect or require different versions. But these kinds of restrictions are unacceptable in a multivendor, distributed, modular system with polymorphic service providers. In other words, yuck.

Thus, service management, extensibility of an architecture, and versioning of services are the problems. Application complexity continues to increase as functionality becomes more and more difficult to extend. Monolithic applications are popular because it is safer and easier to collect all interdependent services and the code that uses those services into one package. Interoperability between applications suffers accordingly because monolithic applications are loath to allow outsiders to access their functionality and thus build a dependence on a certain behavior of a certain version of the code. Because end users demand interoperability, however, software developers are compelled to attempt some integration anyway, but this leads back to the problem of software complexity and completes a vicious cycle of problems that limit the progress of software development. Major yuck.

Component Software: The Breakthrough

Object-oriented programming has long been advanced as a solution to the problems at hand. However, while object-oriented programming is powerful, it has yet to reach its full potential because, in part, no standard framework exists through which software created by different vendors can interact within the same address space—much less across address spaces—and across network and machine architecture boundaries. The major result of the object-oriented programming revolution has been the production of

islands of objects that can't talk to one another across the oceanic boundaries in a meaningful way. Messages in bottles just don't cut it.

The solution is a system in which software developers create *software components*. A software component is reusable pieces of code and data in binary form that can be plugged into other software components from other vendors with relatively little effort. Software components must adhere to an external binary standard, but their internal implementation is completely unconstrained. They can be built using procedural languages as well as object-oriented languages and frameworks (although the latter usually provide many development advantages).

Software component objects are much like integrated circuit (IC) components, and component software is the integrated circuitry of tomorrow. The software industry today is very much where the hardware industry was 20 years ago. At that time, vendors learned how to shrink transistors and put them into a package so that no one had to figure out how to build a particular discrete function—a NAND gate, for example—ever again. Such functions were built into an integrated circuit, a neat package that designers could conveniently buy and design around. As the hardware functions became more complex, the ICs were further integrated to make a board of chips that provided more complex functionality and increased capability. As integrated circuits got smaller yet provided more functionality, boards of chips became just bigger chips. So hardware technology now uses chips to build even bigger chips.

The software industry is now at a point where software developers have been busy building the software equivalent of discrete transistors—software routines—for a long time.

OLE offers a solution and a future—extensible standards and mechanisms to enable software developers to package their functionality, and content, into reusable components, like an integrated circuit. Instead of worrying about how to build functions, developers can simply acquire or purchase that function without having to care about its internal implementation. Just as electronics engineers do not purchase the sources for integrated circuits and rebuild them, OLE allows you to buy a binary component and reuse it, plugging into it through its external interfaces. Not only is this component software a great benefit for developers, but it will eventually allow even end users to assemble custom applications. Users will then have the ability to solve their own problems immediately on a level they understand instead of having to wait months or years for a one-size-fits-all solution. Thus, OLE is the innovation to spark an exciting new fire in all aspects of computing.

OLE Defined

Through its history, there have been many descriptions of OLE, ranging from the sublime to the ridiculous. At some time or another, OLE has been identified with many of its constituent technologies, such as Compound Documents, visual editing, OLE Automation, the Component Object Model, and OLE Controls. You might have seen such narrow identifications in magazines, newsletters, and books.

Each of these definitions is partially true because OLE includes all of them, but each identity misses any conception of a unified whole. A complete definition of OLE must include not only these large and visible technologies but also the minutiae that fill the gaps between them. Furthermore, the definition cannot be rigidly based on the composition of OLE at the time this book was written because Microsoft and other industry groups are continually creating new OLE technologies that are not covered in the pages of this book. Further still, OLE allows software developers to arbitrarily extend the architecture. Therefore, our definition—our framework for understanding OLE—must be flexible to accommodate new additions.

With these characteristics in mind, I define OLE as follows: ***OLE is a unified environment of object-based services with the capability of both customizing those services and arbitrarily extending the architecture through custom services, with the overall purpose of enabling rich integration between components.***

Stated another way, OLE offers an *extensible service architecture,* and in addition to the architecture, OLE itself provides a number of key customizable services, one of which in turn provides for the creation of custom services of any complexity that extend the environment within the same architectural framework. All services, regardless of their complexity, point of storage, point of execution, and implementation, are globally usable by all applications, by the system, and by all other services.

Note that OLE is not a technology for writing every part of an application as the Win32 API is. Where you use the Win32 API to write code and use system resources, you use OLE to share those components with everyone else as well as to access the shared components from the rest of the world.

The collective term used to refer to all services is *component.* To be completely precise, a service is really made of one or more components, but most often a component is a service in itself. In any case, a component is itself made of one or more *objects,* where each object then provides its functionality and content through one or more *interfaces.* These interfaces, in turn, each contain one or more feature-specific member functions. It is through these member functions that one accesses everything a component can do. So

while there may be simple memory-allocation services that have one component, one object, and one interface on that object (with a handful of member functions), other components might have several objects that each have several interfaces through which they expose a great number of features.

Component software is the practical and consumer-oriented realization of the developer-oriented principles of object-oriented programming. Component software is the vision of a computing environment in which developers and end users can incrementally add features to their applications simply by purchasing additional components, rather than by writing such components themselves or by finding more feature-laden monolithic applications.

OLE's life purpose, if you will, is to enable and facilitate component integration and component software. This makes it possible for pieces of applications to talk to one another and makes it possible to create software that involves many pieces of different applications, which is otherwise impossible to do inside a single monolithic program. As a concrete example, consider the creation of a compound document whose contents come from a variety of sources (text, graphics, charts, tables, sound, video, database queries, controls, and so forth). Providing this capability in a nonextensible monolithic application *restricts* the types of information that one can put into a document to only those types that were known to the application at build time. If a new type of content becomes available, as frequently happens, this application would have to be redeveloped, recompiled, and redeployed—a very slow and costly venture—in order to incorporate new content. In contrast, in the component software environment, the application that manages the compound document can allow the user to use content from any available component. If the component environment is designed to be extensible, as OLE is, newly installed components become available immediately to all existing components and applications. Thus, without modification of the document application, a new type of content is usable as soon as the component is installed.

OLE version 1 was created back in 1991, under the now obsolete title "Object Linking and Embedding," for the express purpose of enabling exactly this sort of compound document integration. OLE version 2 was planned originally as performance improvements and enhancements to the functionality of OLE 1, but it grew beyond the boundaries of compound documents into a much more generic service architecture. As we'll see in this chapter and throughout this book, there are many other meaningful and interesting ways to integrate components other than compound documents, such as performing a drag-and-drop operation or controlling an application

programmatically. Certainly compound document technology is still part of OLE and makes a sizable topic for this book, but it no longer enjoys exclusive use of the OLE name. OLE is thus no longer an acronym for Object Linking and Embedding but is rather the name of Microsoft's component integration technology.

You may also notice that OLE is no longer given a version number. The first edition of this book was called *Inside OLE 2,* but this edition is just *Inside OLE.* The reason for this is that OLE 2 implies that there will be an OLE 3. There will not be any such product. *As an extensible service architecture, new features and technologies can be added to OLE within the existing framework without having to change the existing framework!* For example, OLE Controls, a major addition to the architecture, was released more than a year after the original release of OLE 2, but it required no changes to that original technology. Instead, OLE Controls simply builds on and enhances what existed before. This will continue to be true as Microsoft and others add technologies in the future—what exists today will retain its vitality.

Clients and Servers

As mentioned in the previous section, a service or a component is provided through one or more objects, each object consisting of a logical grouping (or isolation) of particular features of the component. These objects make up the communication channels between the user of those objects (some piece of code) and the provider of the objects.

In this book, I use the term *client* to refer to the user of objects—the piece of code that is accessing the functionality and the content of those objects. *Client* literally means "one who uses the services of another," which is exactly what we're talking about. In some cases, I might use the term *user* or *object user* synonymously with *client.*[2] In addition, when a client also maintains the persistent states of object instances, that client is called a *container* because it contains those object instances that are entirely described by the state data. *Container* is most frequently used in the context of compound documents and custom controls.

The provider of a component and its constituent objects is known as a *server,* literally, "one who furnishes services." There is hardly a better term to describe a service provider, for in a programmatic sense a server is the

2. In *Inside OLE 2,* I used the term *user* much more frequently than I do in this edition. That was because the term *client* was poorly understood from the compound document definition that was used with OLE 1, and so in the transition from OLE 1 to OLE 2, *client* was ambiguous. Now that OLE 2 has established itself, it is once again safe to use *client* in the real meaning of the word.

demand-loaded code module—such as a DLL or an EXE—that makes a component and its objects available to the outside world. Without a server, the objects remain hidden from external view. The server holds them out on a silver platter and invites clients to partake of them.

The relationship between a client, a server, and the objects that make up a component or a server is illustrated in Figure 1-1. This is similar to the more general definition of any client-server relationship that might use any number of mechanisms to communicate. In the OLE relationship shown here, however, communication happens through OLE objects.

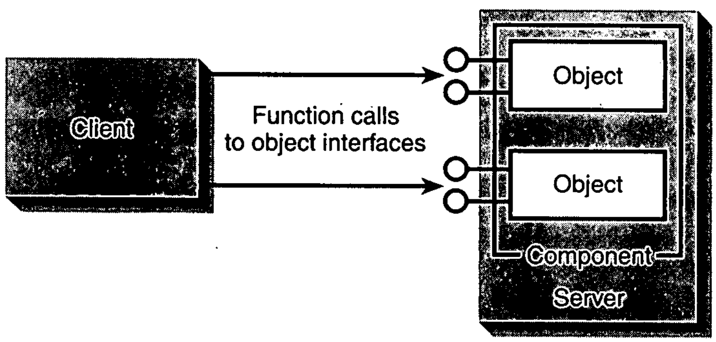


Figure 1-1.
The OLE client-server relationship. A client *uses a component or a service as provided by a* server, *and the communication between client and server happens through OLE objects.*

Object-Based Components: COM

Our definition of OLE describes services as being "object-based." What does this mean exactly? What are "objects"—the communication medium between clients and servers—in the context of OLE? In order to answer this question, we have to look at all the various definitions of the term so we can really understand why an OLE object is what it is. The concepts that form the idea of an OLE object are collectively called the *Component Object Model,* or COM.

Object has to be one of the most bastardized, hackneyed, and confusing terms in the computer industry. Some argue over "real" and "fake" objects. There are academic uses of the term, end-user–oriented uses, and flat out common uses. *Objects* might refer to a methodology, specific techniques in object-oriented programming languages, or icons and other user-interface elements on a computer screen. Then, of course, you have philosophical wackos like me who will argue that the gold-plated Slinky on my desk and the model of the St. Louis Gateway Arch on my bookshelf are also objects.

The Classical Object Definition

When you strip away all the politics, rhetoric, and other baggage and study the original concepts of object-oriented development, you find that an object is an instance of some *class* in which that object is *anything* that supports three fundamental notions:

- *Encapsulation:* all the details about the composition, structure, and internal workings of an object are hidden from clients. The client's view is generically called the object's *interface.* The internals of the object are said to be hidden behind its interface. The interface of the computer keyboard with which I'm typing this book consists of labeled keys, in a specific layout, that I can press to generate a character in my word processor. The internal details of how a keystroke is translated into a character on the screen are encapsulated behind this interface. In the same manner, the internal implementation details of a Smalltalk string object are encapsulated behind the public member functions and variables of that Smalltalk object.
- *Polymorphism:* the ability to view two similar objects through a common interface, thereby eliminating the need to differentiate between the two objects. For example, consider the structure of most writing instruments (pens and pencils). Even though each instrument might have a different ink or lead, a different tip, and a different color, they all share the common interface of how you hold and write with that instrument. All of these objects are polymorphic through that interface—any instrument can be used in the same way as any other instrument that also supports that interface, just as all Slinky toys, regardless of their size and material, act in many ways like any other Slinky. In computer terms, I might have an object that knows how to draw a square and another that knows how to draw a triangle. I can view both of them as having certain features of a "shape" in common, and through that "shape" interface, I can ask either object to draw itself.
- *Inheritance:* a method to express the idea of polymorphism for which the similarities of different classes of objects are described by a common *base class.* The specifics of each object class are defined by a *derived class*—that is, a class derived from the base class. The derived class is said to *inherit* the properties and characteristics of the base class; thus, all classes derived from the same base class are polymorphic through that base class. For example, I might describe a base

class called "Writing Instruments" and make derived classes of "Ballpoint Pen," "Pencil," "Fountain Pen," and so forth. If I wanted to program different "shape" objects, I could define a base class "Shape" and then derive my "Square" and "Triangle" classes from that base class. I achieve polymorphism along with the convenience of being able to centralize all the base class code in one place, within the "Shape" class implementation.

Inheritance is often a sticky point when you come to work with OLE because OLE supports the idea of inheritance only on a conceptual level for the purposes of defining interfaces. In OLE, there is no concept of an object or a class inheriting implementation from another, as there is in C++. But here is the important point: *inheritance is a means to polymorphism and reusability and is not an end in itself.* To implement polymorphic objects in C++, you use inheritance. To create reusable code in C++, you centralize common code in a base class and reuse it through derived classes. But inheritance is not the only means to these two ends! Recognizing this enables us to explore means of polymorphism and reusability that work on the level of binary components, which is OLE's realm, rather than on the level of source code modules, which is the realm of C++ and other object-oriented programming languages.

OLE Objects and OLE Interfaces

As we will explore in this book, objects as expressed in OLE most definitely support the notions of encapsulation, polymorphism, and reusability; again, inheritance is really just a means to the latter two. OLE objects are just as powerful as any other type of object expressed in any programming language, and might be more so. While inheritance and programming languages are excellent ways to achieve polymorphism and reusability of objects or components within a large monolithic application, *OLE is about integration between binary components, and it is therefore targeted at a different set of problems.* OLE is designed to be independent of programming languages, hardware architectures, and other implementation techniques. So there really is no comparison with, and no basis for pitting OLE against, object-oriented programming languages and methodologies; in fact, such languages and methods are very helpful and complementary to OLE in solving customer problems.

The nature of an OLE object, then, is not expressed in how it is implemented internally but rather in how it is exposed externally. As a basis for illustrating the exact structures involved, let's assume we have some software object, written in whatever language (code) that has some properties (data or